1 Introduction

The machine does not isolate us
from the great problems of life
but plunges us more deeply into them.

(Antoine de Saint-Exupéry, Pilot & Writer)

This introductory chapter provides the background and motivation for studying stochastic local search algorithms for combinatorial problems. We start with an introduction to combinatorial problems and present SAT, the satisfiability problem in propositional logic, and TSP, the travelling salesman problem, as the central problems used for illustrative purposes throughout the first part of this book. This is followed by a short introduction to the theory of $\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems. Next, we discuss and compare various fundamental search paradigms, such as the concepts of systematic and local search, after which we formally define and discuss the notion of stochastic local search, one of the practically most important and successful approaches for solving hard combinatorial problems.

1.1 Combinatorial Problems

Combinatorial problems arise in many areas of computer science and other disciplines in which computational methods are applied, such as artificial intelligence, operations research, bioinformatics, or electronic commerce. Prominent examples are tasks such as finding shortest/cheapest round-trips in graphs, finding models of propositional formulae, or determining the 3D-structure of proteins. Other well-known combinatorial problems are encountered in planning, scheduling, time-tabling, resource allocation, code design, hardware design, and genome sequencing. These problems typically involve finding groupings, orderings, or assignments of a discrete set of objects which satisfy certain conditions or constraints. Potential solutions for such problems can be obtained by combining atomic assignments of values to individual objects from the set under consideration. For a scheduling problem, the individual objects could be the events to be scheduled, and their values could be the times at which a given event occurs. This way, typically a huge number of candidate solutions can be obtained; for most combinatorial optimisation problems, the space of potential solutions for a given problem instance is exponential in the size of that instance.

Problems and Solutions

At this point, it is useful to clarify the distinction between problems and problem instances. In this book, by ‘problem’, we mean abstract problems (sometimes also called problem classes), such as ‘for any given set of points in the Euclidian plane, find the shortest round-trip connecting these points’. In this example, an instance of the problem would be to find the shortest round-trip for a specific set of points in the plane. The solution of such a problem instance would be a specific shortest round-trip connecting the given set of points. The solution of the abstract problem however, is a method (or algorithm) which, given a problem instance, determines a solution for that instance. Generally, problems can be defined as sets of problem instances, where each instance is a pair of input data and solution data. This is a elegant mathematical formalisation; however, in this book we will define problems using a slightly less formal, but more intuitive (yet precise) representation.

For instances of combinatorial problems, we draw an important distinction between candidate solutions and solutions. Candidate solutions are potential solutions that may possibly be encountered during an attempt to solve the given problem instance; but unlike solutions, they do not have to satisfy all the conditions from the problem definition. For our shortest round-trip example, typically any valid round-trip connecting the given set of points, regardless of length, would be a candidate solution, while only those candidate round-trips with minimal length would qualify as solutions. It should be noted that while the definition of any combinatorial problem states clearly what is considered a solution for an instance of this problem, the notion of candidate solution is not always uniquely determined by the problem definition, but can already reflect an approach for solving the problem. As an example, consider the variant of the shortest round-trip problem where additionally, we are only interested in trips that visit each given point exactly once. In this case, candidate solutions could be either arbitrary round trips which do not necessarily respect this additional condition, or the notion of candidate solution could be restricted to such round-trips that visit no point more than once.

Decision Problems

Many combinatorial problems can be naturally characterised as decision problems: for these, the solutions of a given instance are characterised by a set of logical conditions. Given a graph and a number of colours, the problem of finding an assignment of colours to its vertices such that two vertices connected by an edge are never assigned the same colour (the Graph Colouring Problem), is an example of a combinatorial decision problem. Other prominent combinatorial decision problems include finding satisfying truth assignments for a given propositional formula (the Propositional Satisfiability Problem, SAT, which we revisit in more detail later in this chapter) or scheduling a series of events such that a given set of precedence constraints is satisfied. For any decision problem, we distinguish two variants:

> the *search variant*, where the goal is, given a problem instance, to find a solution (or to determine that no solution exists);
>
> the *decision variant*, in which for a given problem instance, one wants to answer the question whether or not a solution exists.

These variants are closely related, as clearly, algorithms solving the search variant can always be used to solve the decision variant. Interestingly, for many combinatorial decision problems, the converse also holds: algorithms for solving the decision variant of a problem can be used for finding actual solutions.

Optimisation Problems

Many practically relevant combinatorial problems are optimisation problems rather than decision problems. Optimisation problems can be seen as generalisations of decision problems, where the solutions are additionally evaluated by an *objective function* and the goal is to find solutions with optimal objective function values. For the Graph Colouring Problem mentioned above, a natural optimisation variant exists, where a variable number of colours is used and the goal is, given a graph, to find a colouring of its vertices as described above, using only a minimal (rather than a fixed) number of colours. Any combinatorial optimisation problem can be stated as a maximisation or as a minimisation problem, where often one of the two formulations is more natural. Algorithmically, maximisation and minimisation problems are treated equivalently. For each combinatorial optimisation problem, we distinguish two variants:

> the *search variant*: given a problem instance, find a candidate solution with minimal (or maximal, respectively) objective function value;
>
> the *evaluation variant*: given a problem instance, find the optimal objective function value, *i.e.* the objective function value for an optimal candidate solution.

Clearly, the search variant is the more general of these as with the knowledge of an optimal solution, the evaluation variant can be solved trivially. Additionally, for each optimisation problem, we can define

> *associated decision problems*: given a problem instance and a fixed value b of the objective function, find a candidate solution whose objective function value is smaller than or equal to b (for minimisation problems, greater than or equal to for maximisation problems).

Many combinatorial optimisation problems are defined based on an objective function as well as on logical conditions in such a way that candidate solutions satisfying the logical conditions are called *valid* or *feasible solutions*, and among those, based on the objective function, *optimal solutions* can be distinguished. While this is often a rather natural formalisation, it should be noted that in these cases the logical conditions can always be integrated into the objective function in such a way that feasible solutions correspond to solutions of an associated decision problem (*i.e.*, to candidate solutions exceeding a specific objective function value).

As we will see throughout this book, many algorithms for decision problems can be extended to related optimisation problems in a rather natural way. However, such simple extensions of algorithms which work well on certain decision problems do not always achieve reasonable performance for finding optimal or near-optimal solutions of the related optimisation problem, such that different algorithmic techniques need to be considered for this task.

1.2 Two Prototypical Combinatorial Problems

In the following, we introduce two well-known combinatorial problems which will be used throughout the first part of this book for illustrating algorithmic techniques and approaches. These are the Propositional Satisfiability Problem (SAT), a prominent combinatorial decision problem which plays a central role in several areas of computer science, and the Travelling Salesman Problem (TSP), one of the most extensively studied combinatorial optimisation problems. Besides their prominence and well established role in algorithm development, both problems have the advantage of being conceptually simple, which facilitates the development, analysis, and presentation of algorithms and algorithmic ideas. Both will be discussed in more detail in Part 2 of this book (see Chapters 6 and 8).

The Propositional Satisfiability Problem (SAT)

Roughly speaking, the Propositional Satisfiability Problem is, given a formula in propositional logic, to decide whether there is an assignment of truth values to the propositional variables appearing in this formula, under which the formula evaluates to 'true'. In the following, we first present a formal definition of SAT, and then discuss briefly its computational hardness.

Propositional logic is based on a formal language over an alphabet comprising propositional variables, truth values and logical operators. Using logical operators, propositional variables and truth values are combined into propositional formulae which represent propositional statements. Formally, the syntax of propositional logic can be defined in the following way:

Definition 1.1 (Syntax of Propositional Logic)

$S = V \cup C \cup O \cup \{(,)\}$ is the *alphabet of propositional logic*, with $V = \{x_i \mid i \in \mathbb{N}\}$ denoting the countable infinite set of *propositional variables*, $C = \{\top, \bot\}$ the set of *truth values* (or propositional constants) and $O = \{\neg, \wedge, \vee\}$ the set of *propositional operators*.

The set of *propositional formulae* is characterised by the following inductive definition:

- the truth values $\top$ and $\bot$ are propositional formulae;
- each propositional variable $x_i \in V$ is a propositional formula;
- if F is a propositional formula, then $\neg F$ is also a propositional formula;
- if F_1 and F_2 are propositional formulae, then $(F_1 \wedge F_2)$ and $(F_1 \vee F_2)$ are also propositional formulae. $\square$

Remark: The truth values $\top$ and $\bot$ represent logical constants *true* and *false*, and the operators $\neg, \wedge, \vee$ represent logical negation, conjunction, and disjunction, respectively. Often, additional binary operators, such as '$\leftarrow$' (implication) and '$\leftrightarrow$' (equivalence), are used in propositional formulae. These can be defined based on the operators from Definition 1.1; hence, including them into our propositional language does not increase its expressiveness.

Assignments are mappings from propositional variables to truth values. Using the standard interpretations of the logical operators on truth values, assignments can be used to evaluate propositional formulae. Hence, the semantics of propositional logic can be defined as follows:

Definition 1.2 (Semantics of Propositional Logic)

The *variable set Var*(F) *of formula* F is defined as the set of all the variables appearing in F.

A *variable assignment of formula* F is a mapping $a : \mathit{Var}(F) \mapsto \{\top, \bot\}$ of the variable set of F to the truth values. The set of all possible variable assignments of F is denoted by $\mathit{Assign}(F)$.

The *value* $\mathit{Val}_a(F)$ *of formula* F *under assignment* a is defined inductively based on the syntactic structure of F:

- $\mathit{Val}_a(\top) = \top$
- $\mathit{Val}_a(\bot) = \bot$
- $\mathit{Val}_a(x_i) = a(x_i)$
- $\mathit{Val}_a(\neg F_1) = \neg \mathit{Val}_a(F_1)$
- $\mathit{Val}_a(F_1 \wedge F_2) = \mathit{Val}_a(F_1) \wedge \mathit{Val}_a(F_2)$
- $\mathit{Val}_a(F_1 \vee F_2) = \mathit{Val}_a(F_1) \vee \mathit{Val}_a(F_2)$

The truth values $\top$ and $\bot$ are also known as *true* and *false*, resp.; the operators $\neg$ (*negation*), $\wedge$ (*conjunction*), and $\vee$ (*disjunction*) are defined by the following truth tables:

$\neg$	
$\top$	$\bot$
$\bot$	$\top$

$\wedge$	$\top$	$\bot$
$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

$\vee$	$\top$	$\bot$
$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$

□

Because the variable set of a propositional formula is always finite, the complete set of assignments for a given formula is also finite. More precisely, for a formula containing n variables there are exactly 2^n different variable assignments.

Considering the values of a formula under all possible assignments, the fundamental notion of satisfiability can be defined in the following way:

Definition 1.3 (Satisfiability)

A variable assignment a is a *model of formula* F if and only if $\mathit{Val}_a(F) = \top$; in this case we say that a *satisfies* F.

A formula F is called *satisfiable* if there exists at least one model of F. □

Based on the notion of satisfiability, we can formally define the SAT problem.

Definition 1.4 (The Propositional Satisfiability Problem)

Given a propositional formula F, the *Satisfiability Problem (SAT)* is to decide whether or not F is satisfiable. $\square$

Obviously, SAT can be seen as a combinatorial decision problem, where variable assignments represent candidate solutions and models represent solutions. As for any combinatorial decision problem, we can distinguish a decision variant and a search variant: in the former, only a yes/no decision regarding the satisfiability of the given formula is required; in the latter, also called the model-finding variant, in case the given formula is satisfiable, a model has to be found.

Often, logical problems like SAT are studied for syntactically restricted classes of formulae. Imposing syntactical restrictions usually facilitates theoretical studies and can also be very useful for simplifying the design and analysis of algorithms. Normal forms are syntactically restricted formulae such that for an arbitrary formula F there is always at least one semantically equivalent formula F' in normal form. Thus, each normal form induces a subclass of propositional formulae which is as expressively powerful as full propositional logic. Some of the most commonly used normal forms are introduced in the following definition.

Definition 1.5 (Normal forms)

A *literal* is a propositional variable (called a positive literal) or its negation (called a negative literal). Formulae of the syntactic form $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ are called *conjunctions*, while formulae of the form $d_1 \vee d_2 \vee \ldots \vee d_n$ are called disjunctions.

A propositional formula F is in *conjunctive normal form (CNF)*, if it is a conjunction over disjunctions of literals. The disjunctions are called *clauses*. A CNF formula F is in k-CNF, if all clauses of F contain exactly k literals.

A propositional formula F is in *disjunctive normal form (DNF)*, if it is a disjunction over conjunctions of literals. In this case, the conjunctions are called *clauses*. A DNF formula F is in k-DNF, if all clauses of F contain exactly k literals. $\square$

Example 1.1: A Simple SAT Instance

Let us consider the following propositional formula in CNF:

$$\begin{aligned} F = \quad & (\neg x_1 \vee x_2) \\ & \wedge (\neg x_2 \vee x_1) \\ & \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \\ & \wedge (x_1 \vee x_2) \\ & \wedge (\neg x_4 \vee x_3) \\ & \wedge (\neg x_5 \vee x_3) \end{aligned}$$

For this formula, we obtain the variable set *Var*$(F) = \{x_1, x_2, x_3, x_4, x_5\}$; consequently, there are $2^5 = 32$ different variable assignments. Exactly one of these, $(x_1 = x_2 = \top, x_3 = x_4 = x_5 = \bot)$, is a model, rendering F satisfiable.

The Traveling Salesperson Problem (TSP)

The motivation behind the Traveling Salesperson Problem is the problem faced by a salesperson who needs to visit a number of customers located in different cities and tries to find the shortest round-trip accomplishing this task. In a more general and abstract formulation, the TSP is, given a directed, edge-weighted graph, to find a shortest cyclic path which visits every node in this graph exactly once. In order to define this problem formally, we first introduce the notion of a Hamiltonian cycle:

Definition 1.6 (Path, Hamiltonian cycle)

Let $G = (V, E, w)$ be an edge-weighted, directed graph where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of $n = |V|$ vertices, $E \subseteq V \times V$ the set of (directed) edges, and $w : E \mapsto \mathbb{R}^+$ a function assigning each edge $e \in E$ a weight $w(e)$.

A *path in* G is a list $(u_1, u_2, ..., u_k)$ of vertices $u_i \in V$ ($i = 1, \dots, k$), for which for any pair (u_i, u_{i+1}) ($i = 1, \dots, k-1$) is an edge in G. A *cyclic path in* G is a path for which the first and the last vertex coincide, *i.e.*, $u_1 = u_k$ in the above notation.

A *Hamiltonian cycle in* G is a cyclic path p in G which visits every vertex of G (except for its starting point) exactly once, *i.e.*, $p = (u_1, u_2, \dots, u_n, u_1)$ is a Hamiltonian cycle in G if and only if $n = |V|$, and $\{u_1, u_2, \dots, u_n\} = V$. $\square$

The weight of a path p can then be calculated by adding up the weights of edges in p:

Definition 1.7 (Path weight)

For a given edge-weighted, directed graph and a path $p = (u_1, \dots, u_k)$ in G, the *path weight* $w(p)$ is defined as $w(p) = \sum_{i=1}^{k-1} w((u_i, u_{i+1}))$. $\square$

Now, the TSP can be formally defined in the following way:

Definition 1.8 (The Travelling Salesperson Problem)

Given an edge-weighted, directed graph G, the *Traveling Salesperson Problem (TSP)* is to find a Hamiltonian cycle with minimal path weight in G. $\square$

Often, the TSP is defined in such a way that the underlying graphs are always complete graphs, *i.e.*, any pair of vertices is connected by an edge, because for any TSP instance with an underlying graph G which is not complete, one can always construct a complete graph G' such that the TSP for G' has exactly the same solutions as the one for G. (This is done by making the edge-weights for edges missing in G high enough that these edges can never occur in an optimal solution.) In the remainder of this book we will always assume that TSP instances are specified as complete graphs. Under this assumption, the Hamiltonian cycles in a given graph correspond exactly to the cyclic permutations of the underlying vertex set.

Interesting subclasses of the TSP arise when the edge-weighting function w has specific properties. The following definition covers some commonly used cases:

Definition 1.9 (Asymmetric, symmetric, and Euclidean TSP instances)

A TSP instance is called *symmetric*, if the weight function w of the underlying graph is symmetric, *i.e.*, if for all $v, v' \in V$: $w((v, v')) = w((v', v))$; if w is not symmetric, the instance is called *asymmetric*. The Traveling Salesperson Problem for asymmetric instances is also called Asymmetric TSP (ATSP).

A TSP instance is called *Euclidean*, if the vertices correspond to the points in a Euclidean space and if the weight function w is a Euclidean distance metric. Finally, TSP instances for which the vertices are points on a sphere, and the weight function w represents geographical (Great Circle) distance, are called *geographic*. □

Example 1.2: A Sample (geometric) TSP Instance

Figure 1.1 shows a geometricTSP instance with 22 vertices. The vertices of the underlying graph correspond to 22 locations Ulysses is reported to have visited on his odyssey, and the edge-weights represent the geographic distances between these locations. The figure also shows the optimal solution, *i.e.*, the shortest round-trip (length 7013). This instance can be found as ‘ulysses.22.tsp’ in the TSPLIB Benchmark Library and is attributed to Grötschel and Padberg.

1.3 Computational Complexity

A natural way for solving most combinatorial decision and optimisation problems is, given a problem instance, to search for solutions in the space of its candidate solutions. For that reason, these problems are sometimes

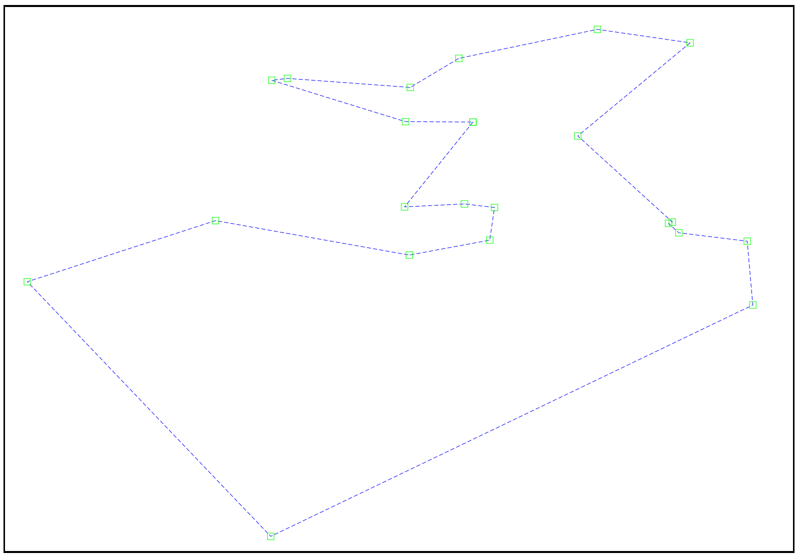

Figure 1.1: A graphic representation of the geographic TSP instance ‘ulysses22’ and its solution; see Example 1.2 for details. **[hh/ts: should add names of locations / replace figure .]**

also characterised as search problems. However, for a given instance of a combinatorial problem, the set of candidate solutions is very large, typically exponential in the size of that instance. For instance, given a SAT instance with 100 variables, typically all 2^{100} different truth assignments are considered candidate solutions. This raises the following question: “Is it possible to search such vast spaces efficiently?” More precisely, we are interested in the time required for solving an instance of a combinatorial problem as a function of the size of this instance.

Questions like this lie at the core of computational complexity theory, a well-established field of computer science with considerable impact on other areas. In the context of this book, complexity theory plays a role, because the primary field of application of stochastic local search algorithms is a class of computationally very hard combinatorial problems, for which no efficient, *i.e.*, polynomial time, algorithms are known. Moreover, to date a majority of the experts in complexity theory believe that for fundamental reasons the existence of efficient algorithms for these problems is impossible.

Complexity of Algorithms and Problems

The complexity of an algorithm is defined on the basis of formal machine models. Usually, these are idealised, yet universal models, designed in a way which facilitates formal reasoning about their behaviour. One of the first, and still maybe the most prominent of these models is the Turing machine. For Turing machines and other formal machine or programming models, computational complexity is defined in terms of the space and time requirements of computations.

Complexity theory usually deals with whole problem classes (generally countable sets of problem instances) instead of single instances. For a given algorithm or machine model, the complexity of a computation is characterised by the functional dependency between the size of an instance and the time and space required to solve this instance. Here, instance size is defined as the length of a reasonably concise description; hence, for a SAT instance, its size corresponds to the length of the propositional formula (written in linear form), while the size of a TSP instance is typically proportional to the size of the underlying graph.

For reasons of analytical tractability, many problems are formulated as decision problems, and time and space complexity are analyzed in terms of the worst-case asymptotic behaviour. Given a suitable definition of the computational complexity of an algorithm for a specific problem, the complexity of the problem itself can be defined as the complexity of the best algorithm for this problem. Because generally time complexity is the more restrictive factor, problems are often categorised into complexity classes with respect to their asymptotic worst-case time complexity.

$\mathcal{NP}$-hard and $\mathcal{NP}$-complete Problems

Two particularly interesting complexity classes are $\mathcal{P}$, the class of problems that can be solved by a *deterministic* machine in polynomial time, and $\mathcal{NP}$, the class of problems which can be solved by a *nondeterministic* machine in polynomial time.[1] Of course, every problem in $\mathcal{P}$ is also contained in $\mathcal{NP}$, basically because deterministic calculations can be emulated on a nondeterministic machine. However, the question whether also $\mathcal{NP} \subseteq \mathcal{P}$, and consequently $\mathcal{P} = \mathcal{NP}$, is one of the most prominent open problems in computer science. Since many extremely application-relevant problems are in $\mathcal{NP}$, but possibly not in $\mathcal{P}$ (*i.e.*, no polynomial time deterministic algorithm is known), this so-called $\mathcal{P} = \mathcal{NP}$-problem is not only of theoretical interest. For these computationally hard problems, the best algorithms known so far have exponential time complexity. Therefore, for growing problem size, the problem instances become quickly intractable, and even the tremendous advances in hardware design have little effect on the size of the problem instances solvable with state-of-the-art technology in reasonable time.

Many of these hard problems from $\mathcal{NP}$ are closely related to each other and can be translated into each other in polynomial deterministic time (these translations are also called *polynomial reductions*). A problem, which is at least as hard as any other problem in $\mathcal{NP}$ (in the sense that each problem in $\mathcal{NP}$ can be polynomially reduced to it) is called *$\mathcal{NP}$-hard*. Thus, $\mathcal{NP}$-hard problems in a certain sense can be regarded as at least as hard as every problem in $\mathcal{NP}$. But they do not necessarily have to belong to the class $\mathcal{NP}$ themselves, as their complexity might be actually higher. $\mathcal{NP}$-hard problems which are contained in $\mathcal{NP}$ are called *$\mathcal{NP}$-complete*; in a certain sense, these problems are the hardest problems in $\mathcal{NP}$.

The SAT problem, introduced in Section 1.2, is the prototypical $\mathcal{NP}$-complete problem. Historically, it was the first problem for which $\mathcal{NP}$-completeness was established [19]. $\mathcal{NP}$-completeness of SAT can directly be proven by encoding the calculations of a Turing machine M for an $\mathcal{NP}$ problem into a propositional formula the models of which correspond to the accepting computations of M. Furthermore, it is quite easy to show that SAT remains $\mathcal{NP}$-complete when restricted to CNF or even 3-CNF formulae [121]. On the other hand, SAT is decidable in linear time for DNF, for 2-CNF [19], and for Horn formulae [30].

Our second example problem, the TSP, is well known to be $\mathcal{NP}$-hard [37]. The same holds for many special cases, such as Euclidean TSPs and

[1]Note that nondeterministic machines are *not* equivalent to machines which make random choices; they are hypothetical machines which can be thought of having the ability to make correct guesses for certain decisions.

even TSPs in which all edge weights are either one or two.

Besides SAT and TSP, many other well-known combinatorial problems are $\mathcal{NP}$-hard or $\mathcal{NP}$-complete, including the Graph Colouring Problem, the Knapsack Problem, as well as many scheduling and timetabling problems, to name just a few [37]. It should be noted that for $\mathcal{NP}$-complete combinatorial decision problems, the search and decision variants are equally hard in the sense that if one could be solved deterministically in polynomial time, the same would apply to the other. This is the case because any algorithm for the search variant also solves the decision variant; and furthermore, given a decision algorithm and a specific problem instance, a solution (if existent) can be constructed by iteratively fixing solution components and deciding solubility of the resulting, modified instance (which requires only a polynomial number of calls to the decision algorithm). In the same sense, for $\mathcal{NP}$-hard optimisation problems, the search and evaluation variants are equally hard. Furthermore, if either of these variants could be solved efficiently (*i.e.*, in polynomial time on a deterministic machine), all decision variants could be solved efficiently as well; and if all decision variants could be solved efficiently, the same would hold for the search and evaluation variant.

One fundamental result of complexity theory states that it suffices to find a polynomial time deterministic algorithm for one single $\mathcal{NP}$-complete problem to prove that $\mathcal{NP} = \mathcal{P}$. This is a consequence of the fact, that all $\mathcal{NP}$-complete problems can be encoded into each other in polynomial time. Today, most computer scientists believe that $\mathcal{P} \neq \mathcal{NP}$; however, so far all efforts of finding a proof for this inequality have been unsuccessful and there is some indication that today's mathematical methods might be too weak to solve this fundamental problem.

Not All Combinatorial Problems are Hard

Although many combinatorial problems are $\mathcal{NP}$-hard, it should be noted that not every computational task which can be formulated as a combinatorial problem is inherently difficult. A well-known example for a problem that, at the first glance, might require searching an exponentially large space of candidate solutions, is the Shortest Path Problem: given an edge-weighted graph G (where all edge-weights are positive) and two vertices u, v in G, find the shortest route from u to v, *i.e.* the path with minimal total edge weight. Fortunately, this shortest path problem can be solved efficiently; in particular, a simple recursive scheme for calculating all pairwise distances between u and any other vertex in the given graph, known as Dijkstra's algorithm [23], can find shortest paths in quadratic time with respect to the number of vertices in the given graph. In general, there are many other combinatorial problems which can be solved by polynomial-time algorithms. In many cases, these efficient algorithms are based on a general method called dynamic programming [9].

Practically Solving Hard Combinatorial Problems

Nevertheless, many practically revelant combinatorial problems, such as scheduling and planning problems, are $\mathcal{NP}$-complete and therefore generally not efficiently solvable to date (and maybe, if $\mathcal{NP} \neq \mathcal{P}$, not efficiently solvable at all). However, being $\mathcal{NP}$-complete or $\mathcal{NP}$-hard does not mean that it is impossible for a problem to be solved efficiently. Practically, there are at least three ways of dealing with these problems:

- Find an application relevant subclass of the problem which can be solved efficiently.
- Use efficient approximation algorithms.
- Use stochastic approaches.

Regarding the first strategy, we have to keep in mind that $\mathcal{NP}$-hardness is a property of a whole problem class P, whereas in practice, often only instances from a certain subclass $P' \subseteq P$ occur. And of course, P' does not have to be $\mathcal{NP}$-hard in general, *i.e.*, while for P an efficient algorithm might not exist, it might still be possible to find an efficient algorithm for the subclass P'; as an example consider the SAT problem for $2-$CNF formulae, which is polynomially solvable.

Furthermore, $\mathcal{NP}$-hardness of a problem is a worst-case complexity result, and typical problem instances might be much easier to solve. Formally, this can be captured in the notion of average case complexity; and although average-case complexity results are typically significantly harder to prove and hence much rarer than worst-case results, empiricial studies suggest that for many $\mathcal{NP}$-hard problems, typical or average case instances can be solved reasonably efficiently. The same applies to the time complexity of concrete algorithms for combinatorial problems; a well-known example is the simplex algorithm for linear optimisation, which has worst-case exponential time complexity, but has been empirically shown to achieve polynomial run-times (w.r.t. problem size) in the average case [150].

If, however, the problem at hand is an optimisation problem which cannot be narrowed down to an efficiently solvable subclass, another option is to accept suboptimal candidate solutions instead of trying to compute optimal solutions. This way, in many cases the computational complexity of the problem can be sufficiently reduced to make the problem practically solvable. In some cases, allowing a comparatively small margin from the optimal solution makes the problem deterministically solvable in polynomial time. In other cases, the approximation problem remains $\mathcal{NP}$-hard, while for practically occurring problem instances, suboptimal solutions of acceptable quality can be found in reasonable time.

For example, it is well known that general TSP instances with arbitrary edge weights are not approximable to a constant factor (approximation ratio) larger than one, *i.e.*, there is no deterministic algorithm which is guaranteed to find solutions within a constant factor of the optimal solution of any given problem instance in polynomial time. Yet, for instances satisfying the triangle inequality, Christofides' polynomial construction algorithm [16] guarantees to return a solution which is at most a factor of 1.5 worse than the optimal solution. Furthermore, in the case of Euclidean TSP instances, a polynomial time approximation scheme exists, *i.e.*, there are algorithms which find solutions for abitrary approximation ratios larger than one in polynomial time w.r.t. instance size [5].

Sometimes, however, even reasonably efficient approximation methods cannot be devised or the problem is a decision problem, to which the notion of approximation cannot be applied at all. In these cases, one further option is to focus on probabilistic rather than deterministic algorithms. At first glance, this idea seems to be appealing: After all, according to the definition of the complexity class $\mathcal{NP}$, at least $\mathcal{NP}$-complete problems can be efficiently solved by (hypothetical) nondeterministic machines. But this, of course, is of little practical use, since it is unlikely that such idealised machines can be built; and for an actual probabilistic algorithm there is merely a chance that it can solve the given problem in polynomial time. In practice, the success probability of such an algorithm can be arbitrarily small. Nevertheless, in numerous occasions, probabilistic algorithms have been found to be considerably more efficient on $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problems than the best deterministic methods available. In other cases, probabilistic methods and deterministic methods complement each other in the sense that for certain types of problem instances one or the other have been found to be superior. SAT and TSP, the two combinatorial problems introduced above, are amongst the most fundamental and best known problems in this category.

Finally, it should be noted that even truly exponential scaling of run-time with instance size does not necessarily rule out solving practically relevant problem instances. For theoretical purposes, complexity analysis typically focusses on asymptotic behaviour, and for exponential scaling, constants (such as the base of the exponential) are mostly not considered. In practice, however, these constants are obviously extremely important, especially, when the size of problem instances that need to be solved has reasonable upper bounds. Consider, for example, an algorithm A with time complexity of $10^{-5} \cdot 1.1^n$ (where n is the problem size), and another algorithm B with time complexity $3 \cdot 10^5 \cdot x^4$. Of course, for big problem instances, here about $n > 510$, A becomes quickly dramatically more costly than B. However, for $n \leq 500$, A is much more efficient than B (for $n = 100$, the performance ratio is larger than 10^{15} in favour of the exponential time algorithm). It is important to keep in mind that exponential complexity should be avoided whenever possible, and does eventually, as instance size grows, make the application of an algorithm infeasible. However, for many problems where exponential time complexity is unavoidable (unless $\mathcal{P} = \mathcal{NP}$), some algorithms, though exponential in time complexity, or even incomplete, can still be dramatically more efficient than others and hence make it feasible to solve the problem for practically interesting instance sizes. This is where heuristic guidance, combined with randomisation and probabilistic decisions (both of which are central issues of this book), can make the difference.

1.4 Search Paradigms

Basically all computational approaches for solving hard combinatorial problems can be characterised as search algorithms. The fundamental idea behind the search approach is to iteratively generate and evaluate candidate solutions; in the case of combinatorial decision problems, evaluating a candidate solution means to decide whether it is an actual solution, while in the case of an optimisation problem, it corresponds to determining the respective value of the objective function. Although for $\mathcal{NP}$-hard combinatorial problems the time complexity of finding solutions can grow exponentially with instance size, evaluating candidate solutions can often be done much more efficiently, *i.e.*, in polynomial time. For example, for a given TSP instance, a candidate solution would correspond to a round-trip visiting each vertex of the given graph exactly once, and its objective function value can be computed easily by summing up the weights associated with all the edges used for that round-trip.

Generally, the evaluation of solution candidates is very dependent on the given problem, and often rather straightforward to implement. The fundamental differences between search algorithms are in the way in which solution candidates are generated, which can have a very significant impact on the algorithms' theoretical properties and practical performance. In this context, general mechanisms can be defined that are applicable to a broad range of search problems. Consequently, in the remainder of this section, we discuss various search paradigms based on their underlying approaches to generating candidate solutions.

Perturbative *vs* Constructive Search

Typically, candidate solutions for instances of combinatorial problems are composed of atomic assignments of values to objects, such as the assignment of truth values to individual propositional variables in the case of SAT. Hence, given candidate solutions can easily be changed into new candidate solutions by modifying one or more of the corresponding atomic assignments. This can be characterised as perturbing a given candidate solution, and hence we classify search algorithms which rely on this mechanism for generating the candidate solutions to be tested as *perturbative search methods*. Applied to SAT, perturbative search would start with one or more complete truth assignments and then in each step generate other truth assignments by changing the truth values of a number of variables in each such assignment.

While for perturbative approaches, the search typically takes place directly in the space of candidate solutions, it can sometimes be useful to also include *partial candidate solutions* in the search space, *i.e.* candidate solutions for which some atomic assignments are not specified. Examples for such partial assignments are partial truth assignments for a SAT instance in which no truth values are specified for certain propositional variables, and partial round-trips for a TSP instance, which correspond to paths in the corresponding graph that visit a subset of the vertices and can be extended into Hamiltionian cycles by adding additional edges.

The task of generating (complete) candidate solutions by iteratively extending partial candidate solution can be formulated as a search problem where typically, the goal is to obtain a ‘good’ candidate solution, where for optimisation problems, the goodness corresponds to a close-to-optimal value of the objective function. Algorithms for solving this type of problem are called *constructive search methods* or *construction heuristics*. As a simple example, consider the following method for generating solution candidates for a given TSP instance: Start at a randomly chosen vertex in the graph, and then iteratively follow an edge with minimal weight connecting the current vertex to one of the vertices that have not yet been visited. This method generates a path that, by adding the starting vertex as a final element to the corresponding list, can be easily extended into a Hamiltonian cycle in the given graph, *i.e.*, a candidate solution for the TSP instance. This simple construction heuristic for the TSP is called the *nearest neighbour heuristic*; on its own, it typically does not generate good solutions (*i.e.* candidate solutions with close-to-optimal objective function values), but it is commonly and successfully used in combination with perturbative search methods (this will be discussed in more detail in Chapter 8).

Systematic *vs* Local Search

A different, and more common, classification of search approaches is based on the distinction between systematic and local search: *Systematic search algorithms* traverse the search space of a problem instance in a systematic manner which guarantees that eventually either a solution is found, or, if no solution exists, this fact is determined with certainty. This typical property of algorithms based on systematic search is called *completeness*. *Local search algorithms*, on the other hand, start at some location of the given search space and subsequently move from the present location to a neighbouring location in the search space, where each location has only a relatively small number of neighbours and each of the moves is determined by a decision based on local knowledge only. Typically, local search algorithms are *incomplete*, *i.e.*, there is no guarantee that an existing solution is eventually found, and the fact that no solution exists can never be determined with certainty. Furthermore, local search methods can visit the same location within the search space more than once. In fact, many local search algorithms are prone to getting stuck in some part of the search space which they cannot escape from without special mechanisms like a complete restart of the search process or some other sort of diversification steps.

As an example for a simple local search method for SAT, consider the following algorithm: Given a propositional formula F in CNF over n propositional variables, randomly pick a variable assignment as a starting point. Then, in each step, check whether the current variable assignment satisfies F. If not, randomly select a variable, and change its truth value from $\perp$ to $\top$ or vice versa. Terminate the search when a model is found, or after a specified number of search steps have been performed unsuccessfully. This algorithm is called *Uninformed Random Walk* and will be revisited in Section 1.5.

To obtain a simple systematic search algorithm for SAT, we modify this local search method in the following way. Given an ordering of the n propositional variables, with each variable assignment a we uniquely associate a number k between 0 and $2^n - 1$ such that digit i of the binary representation of k is 1 if and only if assignment a assigns $\top$ to propositional variable i. Our systematic search algorithm starts with the variable assignment setting all propositional variables to $\perp$, which corresponds to the number 0. Then, in each step we move to the variable assignment obtained by incrementing the numerical value associated with the current assignment by one. The procedure terminates when the current assignment satisfies F or after $2^n - 1$ of these steps. Obviously, this procedure searches the space of all variable assignments in a systematic way and will either return a model of F or terminate unsuccessfully after $2^n - 1$ steps in which case we can be certain that F is unsatisfiable.

Local Search = Perturbative Search?

Local search methods are often, but not always based on perturbative search. The Uninformed Random Walk algorithm for SAT introduced above is a typical example of a perturbative local search algorithm, since in each search step we change the truth value assigned to one variable, which corresponds to a perturbation of a candidate solution. However, local search can also be used for constructive search. This is exemplified by the nearest neighbour heuristic for the TSP introduced earlier in this section, where vertices are iteratively added to a given partial tour based on the weight of the edges leading to vertices adjacent to the last vertex on that tour. Clearly, this process corresponds to a constructive local search on the given graph. Generally, construction heuristics can be interpreted as constructive local search methods, and as we will see in Chapter 2, there are some prominent examples of SLS algorithms based on constructive local search.

In many cases, constructive local search can be combined with perturbative local search. A typical example is the use of nearest neighbour search for generating the starting points for a perturbative local search algorithm for the TSP. Another interesting example is Ant Colony Optimisation [25], which can be seen as a perturbative search, where in each step one or more constructive local searches are performed. This SLS algorithm will be discussed in detail in Section 2.3.

Interestingly, perturbative search, although naturally associated with local search methods, can also provide the basis for systematic search algorithms. As an example, let us consider the systematic variant of the Uninformed Random Walk algorithm for SAT, presented on page 25. The steps of this search algorithm correspond to perturbations of complete variable assignments; consequently, the algorithm can be considered a perturbative systematic search method. As this example shows, perturbative search methods can be complete. It should be noted, however, that we are presently not aware of any perturbative systematic search methods which achieve competitive performance on any hard combinatorial problem.

Constructive Search + Backtracking = Systematic Search

Another interesting relationship can be established between constructive search methods and systematic search algorithms. Let us once more consider our prototypical example for constructive search, the nearest neighbour heuristic for the TSP. If we modify this algorithm such that in each step of the construction process the given partial tour can be extended with arbitrary neigbours of its last vertex, it is clear that the constructive search method thus obtained can in principle find the optimal solution to any given TSP instance. Hence, an algorithm which could systematically enumerate all such constructions would obviously be guaranteed to solve arbitrary TSP instances (given sufficient time), *i.e.*, it would be complete.

Such a complete algorithm for the TSP can be obtained easily by combining the nearest neighbour heuristic with *backtracking*: At each choice point of the construction algorithm (including the initial vertex), a list of all alternative choices is kept. Once a complete tour has been generated, the search process "backtracks" to the most recent choice point at which unexplored alternatives exist, and the constructive search is resumed there using an alternate vertex at this point. This backtracking process first tries alternate choices for recent decisions (which are deep in the corresponding search tree), and once all alternatives are explored for a given choice point, revisits earlier choices; in this latter case, all subsequent choice points are newly generated, *i.e.*, in our example, from that point on, we first use the nearest neighbour heuristic to generate another complete tour, and then recursively continue to revise the choices made in this process.

Visiting all solutions by a backtrack search algorithm leads to an exponential time algorithm which even for very small problems rapidly becomes infeasible. Fortunately, in many situations it is possible to prune large parts of the corresponding search tree which can be shown to not contain any solutions. For example, in the case of the TSP, the search on a given branch can be terminated if the length of the current partial tour plus a lower bound on the length of the completion of the tour exceeds the shortest tour found in the search so far. This type of algorithm is called *branch & bound* or A^* search in the Operations Research and Artificial Intelligence communities, respectively.

For SAT, one can easily devise a backtrack algorithm which searches a binary search tree in which each node corresponds to assigning a truth value to one variable, which is then fixed for the subtree beneath that node. This tree can be pruned considerably by using *unit propagation*, a technique which propagates the logical consequences of particular atomic variable assignments down the search tree and effectively eliminates subtrees from the search which cannot contain a model of the given formula. Unit propagation is one of the key techniques used in all state-of-the-art systematic search algorithms for SAT.

In general, systematic backtracking is a recursive mechanism which can be used to build a complete search algorithm on top of a constructive search method. This approach can be applied to basically any constructive search algorithm. Moreover, many prominent and successful systematic search algorithms, can be decomposed into a constructive search method and some form of backtracking. It should be noted that the construction methods used in this context need not be as "greedy" as the nearest neighbour heuristic. Furthermore, although many well-known systematic search algorithms are deterministic, it is possible to combine randomised construction heuristics with backtracking in order to obtain stochastic systematic search algorithms [50]. There is also some flexibility in the backtracking mechanisms, which does not have to revisit choices in the simple recursive manner indicated above; in fact, as long as there is a reasonably compact representation of all unexplored candidate solutions, basically any strategy that guarantees to eventually evaluate these, leads to a complete search algorithm. In particular, this allows the order in which decisions are revisited to be randomised or dynamically changed based on search progress — approaches which provide the basis for some of the best-known systematic search algorithms for combinatorial problems such as SAT.

Advantages and Disadvantages of Local Search

It might appear that, due to their incompleteness, local search algorithms are generally inferior to systematic methods. But as will be shown later, this is not the case. Firstly, many problems are constructive by nature and it is known that they are soluble. In this situation, the goal of any search algorithm is the generation of a solution rather than just deciding whether a solution does exist. This holds in particular for optimisation problems, like the Traveling Salesperson Problem (TSP) where the actual problem is to find a solution of sufficiently high quality, but also for underconstrained decision problems, which are not uncommon in practice. Obviously, the main advantage of a complete algorithm — its ability to detect that a given problem instance has no solution — is not relevant for finding solutions of soluble instances.

Secondly, in a typical application scenario often the time to find a solution is limited. Examples for such real-time problems can be found in virtually all application domains. Actually one might argue that almost every real-world problem involving interaction with the physical world, including humans, has real-time constraints. Common examples are real-time production scheduling, robot motion planning and decision making, most game playing situations, and speech recognition for natural language interfaces. In these situations, systematic algorithms often have to be aborted after the given time has been exhausted, which — of course — renders them incomplete. This is particularly problematic for certain types of systematic optimisation algorithms which search through spaces of partial solutions without computing complete solutions early in the search (this is the case for many dynamic programming algorithms); if such a systematic algorithm is aborted prematurely, usually no solution candidate is available, while in the same situation local search algorithms typically offer the best solution found so far. Ideally, algorithms for real-time problems should be able to deliver reasonably good solutions at any point during their execution. For optimisation problems this typically means that run-time and solution quality should be positively correlated; for decision problems one could guess a solution when a time-out occurs, where the accuracy of the guess should increase with the run-time of the algorithm. This so-called *any-time property* of algorithms is usually difficult to achieve, but in many situations the local search paradigm is naturally suited for devising any-time algorithms.

As a matter of fact, systematic and local search algorithms are somewhat complementary in their applications. A nice example for this can be found in [74], where a fast local search algorithm is used for finding actual solutions for planning problems the optimality of which is proven by means of a systematic algorithm. As we will discuss later in more detail, different views of the same problem may in certain cases call for local search algorithms, particularly if reasonably good solutions are required within a short time using parallel computation and the knowledge about the problem domain is very limited. In other cases, usually if optimal solutions are required, time constraints are less important and some knowledge about the problem domain can be exploited, systematic search will be the better choice. Finally, there is some evidence that for certain problem classes, local and systematic search methods are most effective on different subclasses of instances. Unfortunately, to date the general question of when to prefer local search over systematic methods and vice versa, remains mainly open.

1.5 Stochastic Local Search

Many widely known and high performance local search algorithms make use of randomised choices in generating or selecting candidate solutions for a given combinatorial problem instance. These algorithms are called *stochastic local search (*or *SLS) algorithms*, and they constitute one of the most successful and widely used approaches for solving hard combinatorial problems.

SLS algorithms have been used for many years in the context of combinatorial optimisation problems. Among the most prominent algorithms of this kind we find the Lin-Kernighan algorithm [82] for the Traveling Salesperson Problem, as well as general methods like Evolutionary Algorithms [6], and Simulated Annealing [77] (these SLS methods will be presented and discussed in Chapter 2). More recently, it has become evident that stochastic local search algorithms can also be very successfully applied to the solution of $\mathcal{NP}$-complete decision problems such as the Graph Colouring Problem (GCP) [60, 97] or the Satisfiability Problem in propositional logic (SAT) [134, 52, 133].

A General Definition of Stochastic Local Search

As outlined in the previous section, local search algorithms generally work in the following way. For a given instance of a combinatorial problem, the search for solutions takes place in the space of candidate solutions, also called the search space. Note that the search space can include partial candidate solutions, as required in the context of constructive search algorithms. The local search process is started by selecting an initial candidate solution, and then proceeds by iteratively moving from one candidate solution to a neighbouring candidate solution, where the decision on each search step is based on a limited amount of local information only. In stochastic local search algorithms, these decisions as well as the search initialisation can be randomised. Formally, a stochastic local search algorithm can be defined in the following way:

Definition 1.10 (Stochastic Local Search Algorithm)

Given a (combinatorial) problem Π, a stochastic local search algorithm for solving an arbitrary problem instance $\pi \in \Pi$ is defined by the following components:

- the *search space* $S(\pi)$ of instance π, which is a set of *candidate solutions* $s \in S$ (also called search positions, locations, configurations, or states);
- a *set of (feasible) solutions* $S'(\pi) \subseteq S(\pi)$;
- a *neighbourhood relation* on $S(\pi)$, $N(\pi) \subseteq S(\pi) \times S(\pi)$;
- an *initialisation function* $init(\pi) : \emptyset \mapsto (S(\pi) \mapsto \mathbb{R})$ which specifies a probability distribution over initial search positions.
- a *step function* $step(\pi) : S(\pi) \mapsto (S(\pi) \mapsto \mathbb{R})$ mapping each position onto a probability distribution over its neighbouring positions, for specifying the local search steps.
- a *termination predicate* $terminate(\pi) : S(\pi) \mapsto (\{\top, \bot\} \mapsto \mathbb{R})$ mapping each position to a probability distribution over truth values ($\top$ = true, $\bot$ = false), which indicates the probability with which the search is to be terminated upon reaching a specific point in the search space.

As an alternative to the initialisation and step functions, one can also specify *initialisation and step procedures* that draw an element from the probability distributions $init(\pi)()$ and $step(\pi)(s)$ (for a given search position s). The same holds for the termination predicate. In the remainder of this book, we will use both types of definitions interchangeably, where $init(\pi)$, $step(\pi, s)$, and $terminate(\pi, s)$ when used in algorithm outlines represent the procedures realising the probabalistic selection from the corresponding probability distributions.

Furthermore, it is often convenient to use $N(s) = \{s' \in S \mid N(s, s')\}$ to denote the set of all points in S which are direct neighbours of a given candidate solution $s \in S$; $N(s)$ is also called the neighbourhood set, or just the neighbourhood of s. $\square$

Remark: In this definition, all components depend on the given problem instance π. Therefore, these could be defined as (higher-order) functions mapping the given problem instance onto the corresponding search space, solution set, *etc.* While this is a straightforward extension of the definition as given above, for increased readability, we specify the components instantiated for a given problem instance; furthermore, we will typically omit the formal reference to the problem instance, by writing S instead of $S(\pi)$, *etc.*

It should be noted that although the mathematical models of the step function and the termination predicate specify as their only parameters the given problem instance and the current search position, many instantiations of these in fact make use of various aspects of search history (such as the number of steps since certain events occured) as well as, in the case of optimisation problems, of the objective function. Furthermore, termination conditions often reflect bounded computational resources (*e.g.*, CPU time) and their choice is therefore more dependent on a specific application context then an integral part of the respective SLS algorithm. In many of the concrete SLS algorithms we present in this book, the choice of a termination predicate will therefore not be discussed.[2]

Based on the components of the definition, the algorithmic outlines in Figures 1.2 and 1.3 specify the semantics of stochastic local search algorithms for the search variants of decision and optimisation problems, respectively. The only major difference between the two versions is that for optimisation problems, the best candidate solution found so far, the so-called *incumbent solution*, is being memorised and returned upon termination of the algorithm (if it is a feasible solution). Furthermore, for decision problems, the termination condition is typically satisfied, if a solution is found, *i.e.* $s \in S'$. For the optimisation algorithm, however, finding a feasible solution $s \in S'$ is typically not a sufficient termination criterion; in fact, many SLS algorithms for optimisation algorithms search through spaces containing feasible solutions only, *i.e.* $S' = S$.

[2]However, issues such as (conditional) search restarts or switches between different search phases, which can be seen as losely related to termination conditions, will be covered in depth later.

```
procedure SLS-Decision
    input problem instance $\pi \in \Pi$
    output solution $s \in S(\pi)$ or $\emptyset$
    $s := init(\pi)$
    while not $terminate(\pi, s)$ do
        $s := step(\pi, s)$
    end
    if $s \in S'(\pi)$ then
        return $s$
    else
        return $\emptyset$
    end
end SLS-Decision
```

Figure 1.2: General outline of a stochastic local search algorithm for a decision problem Π.

Example 1.3: A Simple SLS Algorithm for SAT

For a given problem instance, *i.e.*, a CNF formula F, we define the search space as *Assign*(F), the set of all possible variable assignments of F. Obviously, the set of solutions is then given by the set of all models (satisfying assignments) of F. A frequently used neighbourhood relation is the so-called 'one-flip neighbourhood', which defines two variable assignments to be direct neighbours, if and only if they differ in the truth value of exactly one variable, while agreeing on the assignment of the remaining variables. Formally, this can be written in the following way: For all $a, a' \in \mathit{Assign}(F), N(a, a')$ if and only if there exists $v' \in \mathit{Var}(F)$, such that $\mathit{Val}_a(v') \neq \mathit{Val}_{a'}(v')$ and for all $v \in \mathit{Var}(F) - \{v\} : \mathit{Val}_a(v) = \mathit{Val}_{a'}(v)$.

As an initialisation function, let us consider an 'uninformed' random selection realised by a uniform distribution over the whole search space. This initialisation function randomly selects any assignment of F with equal probability. Formally, it can be written as $init(a) := 1/|S| = 1/2^n$, where $a \in S$ is an arbitrary variable assignment of F and n is the number of variables appearing in F. Analogously, we can define a step function to map

```
procedure SLS-Minimisation
    input problem instance $\pi' \in \Pi'$, objective function $f(\pi')$
    output solution $s \in S(\pi')$ or $\emptyset$
    $s := init(\pi')$
    $\hat{s} := s$
    while not $terminate(\pi', s)$ do
        $s := step(\pi', s)$
        if $f(\pi', s) < f(\pi', \hat{s})$ then
            $\hat{s} := s$
        end
    end
    if $\hat{s} \in S'(\pi')$ then
        return $\hat{s}$
    else
        return $\emptyset$
    end
end SLS-Minimisation
```

Figure 1.3: General outline of a stochastic local search algorithm for a minimisation problem Π' with objective function $f(\pi')$; $\hat{s}$ is the best candidate solution found at any time during the search so far (also called *incumbent solution*).

any variable assignment a to the uniform distribution over all its neighbouring assignments. Formally, if $A' = N(a) = \{a' \in S \mid N(a, a')\}$ is the set of all assignments neighbouring to a, the step function can be defined as $step(a)(a') := 1/|A'|$.

This SLS algorithm is called *uninformed random walk*; as one might imagine, it is quite ineffective, since it does not provide any mechanism for steering the search towards solutions of the problem.

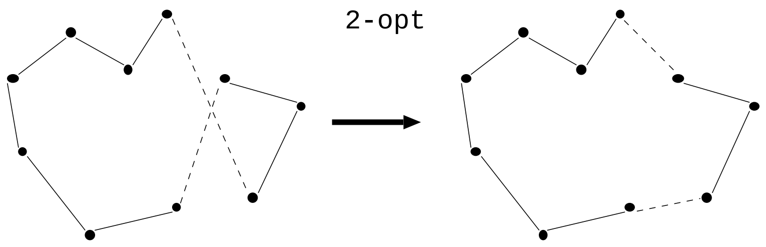

Figure 1.4: Schematic view of a single SLS step based on the standard 2-exchange neighbourhood relation for the TSP.

Neighbourhoods and Neighbourhood Graphs

Generally, the choice of an appropriate neighbourhood relation is crucial for the performance of an SLS algorithm and often, this choice needs to be made in a problem specific way. Nevertheless, there are standard types of neighbourhood relations which form the basis for many successful applications of stochastic local search. One of the most widely used types of neighbourhood relations are the so-called *k-exchange neighbourhoods*, in which two candidate solutions are neighbours if they differ in k solution components. [3]

The neighbourhood used in the simple SAT algorithm from Example 1.5 (as well as in most state-of-the-art SLS algorithms for SAT) is a 1-exchange neighbourhood. For the TSP, one could define a k-exchange neighbourhood such that from a given candidate round-trip, all its direct neighbours can be reached by changing the positions of k vertices in the corresponding permutation. However, this neighbourhood relation was found to be inferior to a different type of k-exchange neighbourhood, where the edges of the given graphs are viewed as the solution components and two candidate round-trips are k-exchange neighbours, if and only if one can be obtained from the other by removing k edges and rewiring the resulting partial tours [120]. Figure 1.4 illustrates two tours which are neighbours under the 2-exchange neighbourhood, a prominent and useful neighbourhood relation for the TSP.

[3]In the literature, k-exchange neighbourhoods are sometimes called k-opt neighbourhoods, since they form the basis for k-opt local search, a simple iterative improvement algorithm, which will be discussed later in this section.

Search Strategies, Steps, and Trajectories

Typically, the first three components of our definition of an SLS algorithm, the search space, solution set, and neighbourhood relation, depend very much on the problem being solved. Together, these components provide the basis for solving a given problem using stochastic local search. But based on a given definition of a search space, solution set, and neighbourhood relation, a wide range of search strategies, specified by the defintion of initialisation and step functions, can be applied. To some extent, such search strategies can be independent from the underlying search space, solution set, and neighbourhood, and consequently can be studied and presented separately from these. In this context, the following concepts are often useful:

Definition 1.11 (Search Steps and Search Trajectories)

Let Π be a (combinatorial) problem, and let $\pi \in \Pi$ be an arbitrary instance of Π. Given an SLS algorithm A for Π according to Definition 1.5, a *search step* (also called *move*) is a pair $(s, s') \in S \times S$ of neighbouring search positions such that the probability for A moving from s to s' is greater than 0, *i.e.*, $N(s, s')$ and $step(s)(s') > 0$.

A *search trajectory* is a finite sequence $(s_0, s_1, \ldots, s_k)$ of search positions s_i $(i = 0, \ldots, k)$ such that for all $i \in \{1, \ldots, k\}$, (s_{i-1}, s_i) is a search step. $\square$

For the simple SLS algorithm for SAT introduced in Example 1.5, each search step is an arbitrary pair of neighbouring variable assignments, and a search trajectory is a sequence of variable assignments in which each pair of successive elements are neighbouring; obviously such a trajectory corresponds to a sequence of search steps. In general, any search trajectory corresponds to a path (also called “walk”) in the neighbourhood graph.

Uninformed SLS: Random Picking and Random Walk

The two (arguably) simplest SLS strategies are Uninformed Random Picking and Uninformed Random Walk. Both are based on an initialisation function which returns the uniform distribution over the entire search space. SLS algorithms based on this initialisation function randomly select any element of the search space S with equal probability for starting the search.

For Uninformed Random Picking, a complete neighbourhood relation is used, *i.e.*, $N = S \times S$, and the step function maps each point in S to a uniform distribution over all its neighbours, *i.e.*, every point in S. Effectively, this strategy randomly samples the search space, drawing a new candidate solution in every step.

Uninformed Random Walk uses the same initialisation function, but for a given, arbitrary neighbourhood relation $N \subseteq S \times S$ its step function returns the uniform distribution over the set of neighbours of the given candidate solution, which implements a uniform, random picking from that neighbourhood in each step. Obviously, for the complete neighbourhood relation, this coincides with Uniform Random Picking; for more restricted neighbourhoods it leads to a strategy that slightly more resembles the intuitive notion of local search.

As one might imagine, both of these uninformed SLS strategies are quite ineffective, since they do not provide any mechanism for steering the search towards solutions. Nevertheless, as we will see later, in combination with more directed search strategies, both Uninformed Random Picking and variants of Uninformed Random Walk play a role for preventing or overcoming premature stagnation in complex and much more effective SLS algorithms.

Evaluation Functions

To improve on the simple uninformed SLS strategies discussed above, a mechanism is needed to guide the search towards solutions. For a given problem instance π, this can be achieved using an *evaluation function* $g(\pi) : S(\pi) \mapsto \mathbb{R}$, which maps each search space position onto a real number in such a way that the global optima correspond to the solutions. Typically, this evaluation function is used for assessing or ranking candidates solutions in the neighbourhood of the current search position. The efficacy of the guidance thus provided depends on properties of the evaluation function and its integration into the search mechanism being used. Typically, the evaluation function is problem specific and its choice is to some degree dependent on the search space, solution set, and neighbourhood underlying the SLS approach under consideration.

In the case of SLS algorithms for combinatorial optimisation problems, the objective function characterising the problem is often used as an evaluation function, such that the values of the evaluation function correspond directly to the quantity to be optimised. However, sometimes different evaluation functions can provide more effective guidance towards high-quality or optimal solutions. For example, this is the case for MAX-SAT, the optimisation variant of SAT which tries to maximise the number of satisfied clauses, where by using a specific evaluation function different from the number of clauses satisfied by a given assignment, local search algorithms with better theoretical approximation guarantees can be obtained [76]. For combinatorial decision problems, sometimes evaluation functions are naturally suggested by the objective functions of optimisation variants, but often there is more than one obvious choice of an evaluation function.

In the case of SLS algorithms for SAT, the following evaluation function g is often used: Given a formula F in CNF and an arbitrary variable assignment a of F, $g(F, a)$ is defined as the number of clauses of F that are unsatisfied under a. Obviously, the models of F correspond to the global minima of g and are characterised by $g(F, a) = 0$. It should be noted that this evaluation corresponds to the objective function of the unweighted MAX-SAT problem, a natural generalisation of SAT which is covered in more detail in Chapter 7.

Remark: In the literature, often no distinction is made between an objective function and an evaluation function. To minimise potential confusion between the definition of the problem to be solved (which, in case of an optimisation problem, includes an objective function) and the definition of SLS algorithms for solving this problem (which might make use of an evaluation function different from the problem's objective function), we systematically distinguish between the two concepts in this book.

Generally, through the use of an evaluation function whose global optima correspond to the (optimal) solutions, decision problems and optimisation problems can be treated analogously. However, for a decision problem, the result of the SLS algorithm is generally useless unless it is a global optimum of the evaluation function and hence corresponds to a solution.

For optimisation problems, suboptimal solutions (usually local minima) can be useful on their own – in which case the respective evaluation function should guide the algorithm to high-quality solutions as effectively as possible (which might complicate or conflict with providing effective guidance towards optimal solutions).

> **Remark:** In the literature, the evaluation function is often treated as an integral part of the definition of an SLS algorithm. Although it is technically possible to define SLS algorithms using the concept of an evaluation function instead of that of a step function, the resulting definitions would capture the concept of stochastic local search less naturally, and would lead to unnecessarily complex or imprecise representations of specific SLS algorithms. These difficulties specifically arise for SLS algorithms which use multiple or dynamically changing evaluation functions (such techniques are prominent and successful in various domains). Using our definition, in many cases the concept of an evaluation function still provides a useful and convenient means for structuring the definition of step functions.

Iterative Improvement

One of the most basic SLS algorithms using an evaluation function is *Iterative Improvement*. Given a search space S, solution set S', neighbourhood relation N, and evaluation function g, Iterative Improvement starts from a randomly selected point in the search space, and then tries to improve the current candidate solution w.r.t. g. The initialisation function is typically the same as in Uninformed Random Picking, *i.e.*, for arbitrary $s \in S$, $init(s) := 1/|S|$. Furthermore, if for a given candidate solution s, $I(s)$ is the set of all neighbouring candidate solutions $s' \in N(s)$ for which $g(s') < g(s)$, then the step function can be formally defined as:

$$step(s)(s') = \begin{cases} 1/|I(s)| & \textit{if } s' \in I(s) \\ 0 & \textit{otherwise} \end{cases}$$

This SLS strategy is also known as *iterative descent* or *hill-climbing*, where the latter name is motivated by the application of Iterative Improvement to maximisation problems. It should be noted that in the case where for a given candidate solution none of its neighbours corresponds to an improvement w.r.t. the evalution function, $step(s)$ is not well-defined. Intuitively, one might imagine that if this case occurs, the search just terminates — an obviously unsatisfying mechanism which we will revisit shortly.

Example 1.4: Iterative Improvement for SAT

Using the same definition for the search space, solution set, and neighbourhood relation as in Example 1.5, we consider the evaluation function g which maps each variable assignment a to the number of clauses of the given formula F which are unsatisfied under a. Iterative Improvement then starts the search at a randomly selected variable assignment (like Uninformed Random Walk, see Example 1.5), and in each step, it randomly selects one of the assignments that leave less clauses unsatisfied than the current candidate solution. Since according to the definition of the neighbourhood relation, each search step corresponds to flipping the truth value associated with one of the variables appearing in F, Iterative Improvement can be seen as always doing such variable flips which increase the overall number of satisfied clauses.

Local Minima

In our definition of Iterative Improvement, the step function was not defined for candidate solutions which do not have any improving neighbours. A candidate solution with this property corresponds to a local minimum of the evaluation function g. Formally, this is captured in the following definition:

Definition 1.12 (Local Minimum, Strict Local Minimum)

Given a search space S, a solution set $S' \subseteq S$, a neighbourhood relation $N \subseteq S \times S$, and an evaluation function $g : S \mapsto \mathbb{R}$, a *local minimum* is a candidate solution $s \in S$ such that for all $s' \in N(s) : f(s) \leq f(s')$. We call a local minimum s a *strict local minimum* if for all $s' \in N(s) : f(s) < f(s')$. (Local maxima and strict local maxima can be defined analogously.) $\square$

Under this definition, solutions which correspond to global minima of the evaluation function, are also considered local minima. Intuitively, local minima, and even more so, strict local minima, are points in the search space, where no search step can achieve an improvement w.r.t. the evaluation function. In cases where an SLS algorithm guided by an evaluation function encounters a local minimum that does not correspond to a solution, this algorithm can "get stuck". This happens, *e.g.*, when an Iterative Improvement algorithm is defined in such a way that it terminates (or just stays at the same candidate solution) when a local optimum is encountered.

In general, there are no (non-trivial) theoretical bounds on the solution quality of local optima for general combinatorial optimisation problems. Guarantees on the solution quality of local optima can only be given for complete neighbourhood relations, in which case any local minimum is also a global minimum. Yet, the size of such complete neighborhoods is exponential w.r.t. instance size and therefore it cannot be searched reasonably efficiently in practice. However, typical instances of combinatorial optimisation problems can be empirically shown to have high quality local optima which often can be found reasonably efficiently by high-performance SLS algorithms.

Computational Complexity of Local Search

While empirically, local minima of basically any instance of a combinatorial optimisation problem can be found reasonably fast, theoretically, in most cases the number of steps needed by an iterative improvement algorithm to find a local optimum cannot be bounded by a polynomial. However, any local search algorithm should at the very least be able to execute individual local search steps efficiently. This idea gives rise to the complexity class $\mathcal{PLS}$ [73]. Intuitively, $\mathcal{PLS}$ is the class of problems for which a feasible local search algorithm exists, in which initial positions and search steps, as well as the evaluation function values of search positions can always be computed in polynomial time (w.r.t. instance size) on a deterministic machine. This means that local optimality can be verified efficiently or, in case a solution is not locally optimal, a neighbouring solution of better quality can be generated in polynomial time. Note that this theoretical concept does not include a statement on the number of local search steps required for reaching a local optimum.

Analogously to the notion of $\mathcal{NP}$-completeness, the class of $\mathcal{PLS}$-complete problems is defined in such a way that it captures the hardest problems in $\mathcal{PLS}$. If for any of these problems local optima can be found in polynomial time, the same would hold for all problems in $\mathcal{PLS}$. It is conjectured that the class of polynomial local search problems is a strict subset of $\mathcal{PLS}$ and hence, in the worst case super-polynomial run-time may be required by any algorithm to find local minima of a $\mathcal{PLS}$-complete problem. The first well-known combinatorial optimisation problem that was shown to be $\mathcal{PLS}$-complete is graph-partitioning of weighted graphs under the Kernighan-Lin neighbourhood [75]. In [111], $\mathcal{PLS}$-completeness has also been shown for the most efficient local search algorithm for the TSP, the Lin-Kernighan heuristic [82]. Furthermore, for the TSP Iterative Improvement using the k-exchange neighbourhood with sufficiently large k, has been shown to be $\mathcal{PLS}$-complete, while the question of $\mathcal{PLS}$-completeness when using 2- or 3-exchange neighbourhoods remains open [72].

Escape Strategies

In many cases, local minima are quite common (this will be further discussed and backed up by actual data in later chapters), and for optimisation problems, they typically do not correspond to reasonably high-quality candidate solutions. Therefore, techniques are needed for avoiding or escaping from local minima. In some sense, such techniques are one of the core issues in SLS algorithm design, and a large number of such escape mechanisms have been proposed and evaluated in the literature. Many of these are discussed in detail or mentioned in passing in the following chapters, and specifically the next chapter introduces some of the most prominent and successful techniques for dealing with the potential problem of getting stuck in local minima. Therefore, we restrict the present discussion to two very simple methods.

One simple way of modifying Iterative Improvement such that local minima are dealt with more reasonably, is to simply start a new search whenever a local minimum is encountered. While this can work reasonably well where the number of local minima is rather small, or restarting the algorithm is not very costly (in terms of overhead cost for initialising data structures, *etc.*), in many cases this technique is very ineffective. Alternatively, one can relax the improvement criterion and, when a local minimum is encountered, randomly select one of the non-improving steps. This can be realised as a uniform random selection among all neighbours of the current search position (which corresponds to an Uninformed Random Walk step), or it can be done by randomly selecting one of the neighbours which give the lowest increase in evaluation function value (this corresponds to a mildest ascent step and is closely related to a variant of Iterative Improvement which will be discussed in more detail in Chapter 2).

In either case, it cannot be guaranteed that the search algorithm effectively escapes from arbitrary local minima, because the nature of a local minimum can be such that after any such "escape step", the only improving step available leads directly back into the same local minimum. Furthermore, in the case of local minima which are not strict, best improvement escape steps will lead to walks in regions of neighbouring candidate solutions with identical evaluation function values, which are often called *plateaus*. As we will see later, such plateaus can be very extensive, and it can be difficult to decide whether the search is stuck in a plateau that does not allow any further improvement without an effective escape mechanism.

Intensification *vs* Diversification

As we will show in more detail in later chapters of this book, the strong randomisation of local search algorithms, *i.e.*, the utilisation of stochastic choice as an integral part of the search process, can significantly increase their performance and robustness. However, with this potential comes the need to balance randomised and goal-directed components of the search strategy, a trade-off which is often characterised as "diversification *vs* intensification". "Intensification" refers to a search strategy which aims to greedily improve solution quality or the chances of finding a solution in the immediate future by exploiting, for instance, the guidance given by the evaluation function. "Diversification" strategies try to prevent search stagnation by making sure that the search process achieves a reasonable coverage when exploring the search space, and does not get stuck in relatively confined regions in which at some point no further progress can be made. In this sense, Iterative Improvement is an intensification strategy, while Uninformed Random Walk is a diversification strategy, and as we will see in the next chapter, both strategies can be combined into an SLS approach called *Iterative Improvement with Random Walk* which typically shows improved performance over both pure search methods.

A large variety of techniques for combining and balancing intensification and diversification strategies have been proposed, and to some extent these will be presented and discussed in the remainder of this book. While the resulting SLS algorithms often perform very well in practice, typically their behaviour is not well understood. The successful application of these algorithms is often based on intuition and experience rather than on theoretically or empirically derived principles and insights, particularly when it comes to the trade-off between diversification and intensification. While in this context, problem specific knowledge is often (if not typically) crucial for achieving peak performance and robustness, a solid understanding of the SLS strategies and algorithms currently available, combined with detailed knowledge of their properties and characteristics is at least of equal importance. The latter is especially relevant in cases where one of the reasons for applying SLS algorithms is a lack of specfic knowledge about the problem to be solved; in this situation, where specialised algorithms are typically not available, SLS algorithms are attractive because they often allow solving the problem reasonably efficiently using fairly generic and easily implemented techniques. More importantly, for many hard combinatorial problems, such generic SLS techniques can also quite be naturally extended with or adapted based on problem-specific knowledge as it becomes available. The specialised SLS algorithms thus obtained are often amongst the best-known techniques for solving these problems, and specifically for large instances of optimisation problems or under tight constraints on time and other computational resources, in many cases represent the only known methods for finding solutions in practice.

In the following chapters, we will introduce and discuss a broad range of SLS algorithms, covering many state-of-the-art generic SLS techniques. Our discussion will focus on underlying general properties and design principles, such as the combination of search strategies and methods for balancing intensification and diversification aspects of search. Later, we will show in detail how these general techniques are applied and adapted to specific combinatorial problems, yielding high-performing, state-of-the-art algorithms for solving these problems.

1.6 Further Readings and Related Work

Due to the introductory nature of this chapter, there is a huge body of literature related to the concepts presented here. Introductions to combinatorial problems and search methods can be found in many modern or classic textbooks on combinatorial optimization, Operations Research or Artificial Intelligence (such as [2, 80, 107, 113, 116, 119, 117, 127], *etc.*); for details on heuristic search, see also [115]. For a slightly different definition of combinatorial optimisation problems we refer to the classical text by Papadimitriou and Steiglitz [113]. A detailed discussion of complexity theory, $\mathcal{NP}$-completeness, and $\mathcal{NP}$-hard problems can be found in [37, 112] or [121].

For a general reference to recent research on the Propositional Satisfiability Problem we refer to the book edited by van Maaren, Gent, and Walsh [39] and to the overview article by Gu et al. [**?**]. For details and a large number of variants of the TSP we refer to the now classical book edited by Lawler et al. [81] or the monograph by Reinelt [120]. For a detailed account of the state-of-the-art in TSP solving with SLS algorithms up to 1997, the book chapter by Johnson and McGeoch [72] is the best reference; results of more recent variants are collected on the web pages for the 8th DIMACS Challenge on the TSP. Regarding stochastic local search methods for SAT, early studies are [134] and [52], while some of the better performing algorithms are presented in [90]. For an overview and comparison of the best performing SLS algorithms for SAT up to 2000 we refer to [68].

1.7 Summary

This chapter started with a brief introduction to combinatorial problems and distinguished between two main types of problems, decision and optimisation problems. We introduced the Propositional Satisfiability Problem (SAT) and the Travelling Salesman Problem (TSP) as two prototypical combinatorial problems. Both problems are conceptually simple and easy to state, which facilitates the design and analysis of algorithms. At the same time, they are computationally hard and appear at the core of many real-world applications; hence, these problems pose a constant challenge for the development of new algorithmic techniques for solving hard combinatorial problems. Many combinatorial problems, including SAT and TSP, are $\mathcal{NP}$-hard; consequently, there is little hope for finding algorithms with better than exponential worst-case behaviour. However, this does not imply that all instances of these problems are intrinsically hard. Interesting or application relevant subclasses of hard combinatorial problems can be efficiently solvable. For example, for many optimisation problems, efficient approximation algorithms exist which can find good solutions reasonably efficient. Additionally, stochastic algorithms can help in solving combinatorial problems more robustly and efficiently in practice.

In Section 1.4, we discussed various search paradigms and highlighted their relations and properties. For example, we distinguished perturbative local search methods which operate on fully instantiated candidate solutions, and constructive search algorithms, which iteratively extend partial candidate solutions. Constructive search algorithms can always be combined with backtracking, leading to complete, systematic search methods which are traditionally known as tree search or refinement search techniques. Many local search algorithms have the advantage of being easily applicable to a broad range of combinatorial problems, for many of which they have been shown to be the most effective solution methods. Furthermore, they are typically rather easy to implement and often they have attractive any-time properties. But these advantages come at a price. Local search algorithms are typically incomplete and, particularly in the case of stochastic local search methods, they are generally difficult to analyse – an issue that will be addressed in more detail in Chapter 4.

Finally, in Section 1.5, we gave a general definition of stochastic local search algorithms which covers both perturbative as well as constructive methods within a unified framework. Based on this definition we introduced and discussed a number of simple SLS algorithms such as Iterative Improvement, which forms the basis of many of the more complex SLS techniques presented in the next chapter.

1.8 Exercises

Exercise 1.1 (Easy) Consider the following *graph colouring problem*: Given a graph $G = (V, E)$ with vertex set V and edge relation E, the graph colouring problem assign colours $c_1, c_2, \ldots, c_k$ to the vertices such that two vertices which are connected by an edge in E are never assigned the same colour. Show how this problem fits the definition of a combinatorial problem and state the different decision and optimisation variants as defined in Section 1.1.

Exercise 1.2 (Hard) Consider the problem of finding a Hamiltonian cycle in a given (undirected) graph. Is this Hamiltonian cycle problem $\mathcal{NP}$-hard? (Hint: Think about relations between this problem and the TSP.)

Exercise 1.3 (Hard) Consider the following argument. For Euclidean TSPs, polynomial algorithms exist for arbitrary approximation ratios. Hence, the associated decision problems can be solved in polynomial time for arbitrary solution quality bounds, which implies that the search variant is also efficiently solvable. Why is this argument flawed? (Hint: Think carefully about the nature of the solution quality bounds.)

Exercise 1.4 (Easy) Consider the following recursive algorithm for SAT:

```
procedure DP-SAT($F$,$A$)
    input propositional formula $F$, partial truth assignment $A$
    output true or false
    if $A$ satisfies $F$ then
        return true
    end
    if $\exists$ unassigned variable in $A$ then
        randomly select variable $x$ which is unassigned in $A$
        $A'$ := $A$ extended by $x := \top$
        $A''$ := $A$ extended by $x := \bot$
    if DP-SAT ($F$,$A'$) = true or DP-SAT ($F$,$A''$) = true then
        return true
    else
        return false
    end
end DP-SAT
```

Which search paradigm does this algorithm implement and which of the properties discussed in Section 1.4 does it have?

Exercise 1.5 (Medium) Design a complete stochastic local search algorithm for SAT (it can be very simple and naive) and show how it fits in the formal definition from Section 1.5. Show also that your algorithm is complete and discuss the practical impact of its completeness.

Exercise 1.6 (Medium) Consider the following, alternative definition of a stochastic local search algorithm.

> Given a (combinatorial) problem Π, a stochastic local search algorithm for solving an arbitrary problem instance $\pi \in \Pi$ is defined by the following components:
>
> - a (directed) *search graph* $G(\pi) = (V, E)$, where the elements V are the candidate solutions of π and the arcs in E connect any candidate solution to those candidate solutions which can be reached in one search step;
> - an *evaluation function* f_π which assigns a numerical value $f_\pi(s)$ to each candidate solution s and whose global maxima correspond to the (optimal) solutions of π;
> - an *initialisation procedure* $init(\pi)$, which determines a candidate solution at which the search process is started;
> - a *iteration procedure* $iter(\pi)$, which for any candidate solution s selects a candidate solutions s' such that $(s, s') \in E$;
> - a *termination function* $terminate(\pi)$ which for a given candidate solution determines whether the search is to be terminated (this function can make use of a random number generator and a limited amount of memory on earlier events in the search process).

Is this definition equivalent to the one given in Section 1.5 *i.e.*, does it cover the same class of algorithms? Discuss the differences between the definitions and try to decide which one is better.